CAPITOLO 10

Il package delle utilità

di Michael Girdley e Richard Lesh, rivisto da Billy Barron

IN QUESTO CAPITOLO

Questo capitolo descrive le classi del package `java.util` di Java. Queste classi contengono molte funzionalità, della cui implementazione si occupa di norma il programmatore. Spesso i programmatori ritengono che sarebbe tutto molto più semplice se esistesse un oggetto integrato che eseguisse "alcuni compiti comuni, ma complicati". Il package `java.util` è un tentativo ben progettato ed efficace di soddisfare molte di queste esigenze specializzate.

In molti linguaggi a volte si rende necessario implementare uno stack o la classe di una tabella di hash e tutti i metodi corrispondenti; in Java vi è un tipo di stack integrato che permette di includere in modo veloce ed efficiente le proprie strutture di dati dello stack nei programmi di Java. In questo modo ci si può concentrare su problemi di progettazione e di implementazione più importanti. Queste classi sono utili in numerose situazioni e sono i componenti fondamentali delle strutture di dati più complicate utilizzate in altri package di Java e nelle applicazioni che vengono sviluppate.

Questo capitolo discute i seguenti argomenti.

- Tutte le funzionalità del package delle utilità.
- L'implementazione della maggior parte delle classi nel package delle utilità.

Se non è diversamente specificato, tutte le interfacce e le classi discusse in questo capitolo estendono la classe `java.lang.Object`.

Nella Tabella 10.1 sono indicate le classi che compongono il package delle utilità e che vengono discusse in questo capitolo.

Tabella 10.1 *Classi importanti nel package delle utilità.*

Classe	*Descrizione*
`BitSet`	Implementa una collezione di valori binari.
`Calendar`	Utilizzata per implementare un calendario.
`Date`	Utilizzata per memorizzare e utilizzare i dati di data e ora.
`Dictionary`	Utilizzata per memorizzare una serie di coppie chiave-valore.
`GregorianCalendar`	Utilizzata per implementare un calendario gregoriano.
`Hashtable`	Utilizzata per memorizzare una tabella di hash.
`Locale`	Utilizzata per implementare una località.
`Observable`	Utilizzata per memorizzare dati osservabili.
`Properties`	Utilizzata per memorizzare un elenco di proprietà.
`Random`	Utilizzata per generare un numero pseudo-casuale.
`SimpleTimeZone`	Utilizzata per implementare un fuso orario semplificato.
`Stack`	Utilizzata per memorizzare e implementare uno stack.
`StringTokenizer`	Utilizzata per suddividere in token una stringa.
`TimeZone`	Utilizzata per memorizzare informazioni su un fuso orario.
`Vector`	Utilizzata per memorizzare un tipo di dati vettore.

Per chi non ha familiarità con alcuni di questi tipi di dati, la classe `Dictionary` viene utilizzata per implementare un dizionario in un programma. Una `Hashtable` è un tipo di dati in memoria, in cui la ricerca risulta molto più veloce rispetto a quella nelle altre strutture di dati, in quanto i dati sono memorizzati sulla base di una chiave derivata da una determinata formula. Uno `Stack`, naturalmente, funziona come se si impilassero i dati uno sopra l'altro, in un'unica pila; le uniche due manipolazioni che si possono apportare allo stack consistono nel rimuovere l'elemento superiore o nell'inserire nella parte superiore un altro elemento. La classe `Vector` implementa una struttura di dati interessante in grado di iniziare con una capacità limitata e di modificare successivamente le dimensioni in modo da contenere i dati che vi si inseriscono. La classe `Vector` può essere considerata come un "array che può crescere".

Elenchi collegati, code, strutture di ricerca e altre strutture di dati dinamiche

Si potrebbe pensare che la classe `Vector` elimini la necessità di creare le proprie strutture di dati. A volte però si desidera conservare lo spazio al massimo o accedere ai dati in modo specializzato; in questi casi, esiste una tecnica per implementare in Java queste strutture di dati.

Come si sa, in Java non esistono puntatori. Dato che gli elenchi a collegamento dinamico e le code sono implementati utilizzando i puntatori, non è possibile creare direttamente queste due strutture di dati in Java.

Come per diversi altri compiti, è necessario eseguire dei passaggi particolari, in quanto l'implementazione degli elenchi, delle code e di altre strutture di dati dinamiche non è intuitiva.

Per definire una struttura di dati personalizzata, è necessario sfruttare il fatto che i riferimenti agli oggetti in Java sono già dinamici, come dimostrato e richiesto dalle procedure utilizzate in Java, ad esempio le interfacce e le implementazioni astratte.

Se si è abituati a implementare elenchi dinamici o code in C++, il formato utilizzato in Java per creare la propria versione di queste strutture dovrebbe essere familiare. Ad esempio, il seguente codice crea una classe `Nodo` per l'elenco che contiene una stringa:

```
class Nodo {
    String Nome;
    Nodo Prec;
    Nodo Succ;
}
```

Naturalmente questo codice crea un *elenco a doppio collegamento*, che ha collegamenti da e per altri nodi che contengono stringhe. Altrettanto facile risulta la conversione di questo tipo per collegare oggetti in qualsiasi modo e ottenere qualsiasi comportamento desiderato: code, stack (si ricordi che nella libreria di classi esiste già un oggetto `Stack`), elenchi a doppio collegamento, elenchi circolari, strutture di ricerca binarie e così via.

Per implementare un tale elenco, si crea una classe `DoppioElenco` che contiene un oggetto `Nodo` e collegamenti in uscita. È possibile utilizzare la parola chiave `null` per rappresentare un oggetto vuoto. Di seguito viene mostrato un esempio della dichiarazione di `DoppioElenco`:

```
class DoppioElenco {
    // Dichiara l'intestazione dell'elenco come tipo Nodo creato prima.
    // Inoltre, la imposta a un oggetto vuoto.
    Nodo TestaElenco = null;
.
.
}
```

È quindi possibile creare metodi che agiscono sull'elenco, ad esempio `InserisciNodo()` o `RipulisciElenco()`, oppure creare un metodo costruttore per la classe `Nodo` che accetta parametri per impostare i nodi precedenti e successivi al momento della creazione, o un metodo quale `SetNext()` o `SetNextToNull()`.

Oltre a tutto ciò, vi è un'altra sorpresa: non è necessario preoccuparsi di liberare lo spazio allocato per creare i nodi, in quanto i processi di garbage collection di Java si occupano di ciò. È sufficiente creare gli oggetti quando serve e Java se ne prende cura.

Utilizzo del package delle utilità

Il package delle utilità ha tre interfacce, ma di norma se ne utilizzano solo due: `Enumeration` e `Observer`. L'altra interfaccia è `EventListener`, che viene comunemente utilizzata internamente all'AWT per gestire gli eventi; i programmatori la implementano solo raramente. Un'*interfaccia* è una serie di metodi che devono essere scritti per tutte le classi che devono implementarla. In questo modo è possibile utilizzare in modo uniforme tutte le classi che implementano l'interfaccia. Le interfacce `Enumeration` e `Observer` possono essere descritte nel seguente modo.

- ✔ `Enumeration`. Interfaccia per classi che possono enumerare un vettore.
- ✔ `Observer`. Interfaccia per classi che possono osservare oggetti osservabili.

L'interfaccia `Enumeration` viene utilizzata per le classi che possono richiamare dati da un elenco, elemento per elemento. Ad esempio, esiste una classe `Enumeration` nel package delle utilità che implementa l'interfaccia `Enumeration` per l'utilizzo con la classe `Vector`. In questo modo non serve un'analisi dettagliata delle diverse classi di strutture di dati. L'interfaccia `Observer` è utile per progettare classi che possono osservare i cambiamenti che avvengono in altre classi.

Alcuni esempi in questo capitolo non sono applet, ma applicazioni. Molte di queste strutture di dati possono essere spiegate meglio con semplice testo di input e di output. Eliminando tutto ciò che è collegato agli applet, gli esempi vengono semplificati e gli argomenti trattati risultano più chiari.
Quando si utilizza un codice incluso in questo capitolo in un'applet, è necessario ricordare che alcuni esempi non sono veri applet e pertanto è necessario tenere conto delle differenze tra questi e le applicazioni.

L'interfaccia Enumeration

L'interfaccia `Enumeration` specifica una serie di metodi utilizzati per *enumerare*, vale a dire iterare, un elenco. Un oggetto che implementa questa interfaccia può essere utilizzato per iterare un elenco una volta sola, in quanto l'oggetto `Enumeration` viene consumato dall'utilizzo.

Ad esempio, un oggetto `Enumeration` può essere utilizzato per stampare tutti gli elementi di un oggetto `Vector`, `v`, come nel seguente esempio:

```
for (Enumeration e=v.elements();e.hasMoreElements();)
   System.out.print(e.nextElement()+" ");
```

L'interfaccia `Enumeration` specifica solo due metodi: `hasMoreElements()` e `nextElement()`. Il metodo `hasMoreElements()` deve restituire `true` se vi sono altri elementi nella enumerazione, mentre il metodo `nextElement()` deve restituire un oggetto che rappresenta l'elemento successivo all'interno dell'oggetto che viene enumerato. I dettagli di come viene implementata l'interfaccia `Enumeration` e di come vengono rappresentati internamente i dati vengono gestiti dall'implementazione della classe specifica.

L'interfaccia Observer

L'interfaccia `Observer`, se implementata da una classe, permette a un oggetto della stessa di osservare altri oggetti della classe `Observable`. L'interfaccia `Observer` viene informata ogni volta che l'oggetto `Observable` sotto osservazione subisce un cambiamento.

L'interfaccia specifica un solo metodo, `update(Observable, Object)`, che viene richiamato dall'oggetto osservato per informare l'`Observer` di un cambiamento. Assieme a qualsiasi altro oggetto che l'oggetto osservato desidera passare all'`Observer`, viene passato un riferimento all'oggetto osservato. Il primo argomento permette all'`Observer` di operare sull'oggetto osservato, mentre il secondo argomento viene utilizzato per passare le informazioni dall'oggetto osservato all'`Observer`.

Classi

Il package delle utilità contiene molte classi diverse che forniscono numerose funzionalità. Nonostante queste classi in genere non abbiano molto in comune, tutte forniscono il supporto per le strutture di dati più comuni utilizzate dai programmatori. Le tecniche descritte nei seguenti paragrafi permettono di creare classi specializzate per sopperire a quelle che mancano nel package.

Le classi incluse nel package `java.util`, per quanto limitate, offrono comunque un grande vantaggio rispetto agli altri linguaggi, semplificando alcune cose ed eliminando buona parte dei compiti inutili che dovevano essere eseguiti in passato, per quanto concerne il liberare memoria e l'esecuzione di compiti di programmazione ordinari.

Vi sono tuttavia alcune limitazioni. Ad esempio, per implementare alcune delle strutture di dati più complicate è necessario eseguire delle operazioni particolari, e la velocità non è particolarmente elevata. Java offre una combinazione di potenza e semplicità, sacrificando la velocità. Non ci si deve tuttavia preoccupare che i programmi siano troppo lenti; nonostante Java non sia efficiente come C++ e C, batte comunque Visual Basic in termini di dimensioni e di velocità.

La classe BitSet

La classe `BitSet` implementa un tipo di dati che rappresenta una collezione di bit, che cresce dinamicamente a mano a mano che sono necessari più bit. Questa classe è utile per rappresentare una serie di valori `true` e `false`. I bit specifici vengono identificati utilizzando interi non negativi. Il primo bit è il bit `0`.

La classe `BitSet` si rivela maggiormente utile per memorizzare un gruppo di valori `true/false` correlati, ad esempio le risposte Sì e No di un utente. Ad esempio, se un applet contiene diversi pulsanti di opzione, è possibile inserire questi valori in un'istanza della classe `BitSet`. La classe è utile anche per le mappe di immagini grafiche. È possibile creare serie di bit che rappresentano un pixel alla volta, anche se per questo scopo è più semplice utilizzare la classe `Graphics`.

I singoli bit di una serie vengono attivati o disattivati per mezzo dei metodi `set()` e `clear()`, mentre per mezzo del metodo `get()` si richiedono informazioni. Tutti questi metodi utilizzano come unico argomento il numero di bit specifico. È possibile eseguire le operazioni booleane di base `AND`, `OR` e `XOR` su due serie di bit utilizzando i metodi `and()`, `or()` e `xor()`. Poiché questi metodi modificano una delle serie, generalmente si utilizza il metodo `clone()` per creare un duplicato di una serie di bit, quindi si esegue l'operazione `AND`, `OR` o `XOR` sul clone con la seconda serie di bit. Il risultato dell'operazione finisce nella serie di bit clonata. Il programma `BitSet1` nel Listato 10.1 mostra le operazioni di base della classe `BitSet`.

Listato 10.1 *BitSet1.java: un programma di esempio di BitSet.*

```
import java.io.DataInputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.BitSet;
class BitSet1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        BufferedReader dis=new BufferedReader(new InputStreamReader(System.in));
        String bitstring;
        BitSet set1,set2,set3;
        set1=new BitSet();
        set2=new BitSet();
        // Ottiene la prima serie di bit e la memorizza
        System.out.println("Serie di bit #1:");
        bitstring=dis.readLine();
        for (short i=0;i<bitstring.length();i++){
            if (bitstring.charAt(i)=='1')
                set1.set(i);
            else
                set1.clear(i);
        }
        // Ottiene la seconda serie di bit e la memorizza
        System.out.println("Serie di bit #2:");
        bitstring=dis.readLine();
        for (short i=0;i<bitstring.length();i++){
            if (bitstring.charAt(i)=='1')
                set2.set(i);
            else
                set2.clear(i);
        }
        System.out.println("BitSet #1: "+set1);
        System.out.println("BitSet #2: "+set2);
        // Test dell'operazione AND
        set3=(BitSet)set1.clone();
        set3.and(set2);
        System.out.println("set1 AND set2: "+set3);
        // Test dell'operazione OR
        set3=(BitSet)set1.clone();
        set3.or(set2);
        System.out.println("set1 OR set2: "+set3);
        // Test dell'operazione XOR
```

```
        set3=(BitSet)set1.clone();
        set3.xor(set2);
        System.out.println("set1 XOR set2: "+set3);
    }
}
```

L'output di questo programma è:

```
Serie di bit #1:
1010
Serie di bit #2:
1100
BitSet #1: {0, 2}
BitSet #2: {0, 1}
set1 AND set2: {0}
set1 OR set2: {0, 1, 2}
set1 XOR set2: {1, 2}
```

Nella Tabella 10.2 sono riepilogati tutti i metodi disponibili in `BitSet`.

Tabella 10.2 *I metodi di BitSet.*

Metodo	*Descrizione*
Costruttori	
`BitSet()`	Crea un `BitSet` vuoto.
`BitSet(int)`	Crea un `BitSet` vuoto di dimensioni predefinite.
Metodi	
`and(BitSet)`	Esegue l'AND logico della serie di bit di un oggetto con un altro oggetto `BitSet`.
`clear(int)`	Elimina un bit specifico.
`clone()`	Crea un clone dell'oggetto `BitSet`.
`equals(Object)`	Confronta un oggetto con un altro oggetto `BitSet`.
`get(int)`	Restituisce il valore di un bit specifico.
`hashCode()`	Restituisce il codice di hash.
`or(BitSet)`	Esegue l'OR logico della serie di bit di un oggetto con un altro oggetto `BitSet`.
`set(int)`	Imposta un bit specifico.
`size()`	Restituisce le dimensioni di una serie.
`toString()`	Converte i valori dei bit in stringa.
`xor(BitSet)`	Esegue l'XOR logico della serie di bit di un oggetto con un altro oggetto `BitSet`.

Oltre a estendere la classe `java.lang.Object`, `BitSet` implementa l'interfaccia `java.lang.Cloneable`. Naturalmente, questo permette di clonare le istanze di un oggetto in modo da creare un'altra istanza della classe.

La classe Calendar

`Calendar` è una classe astratta utilizzata per la conversione delle date. È possibile utilizzarla per convertire un oggetto `Date` in campi, ad esempio `YEAR`, `MONTH`, `HOUR` e così via, oppure utilizzare questi campi per aggiornare un oggetto `Date`.

Nella definizione dell'API esiste una sola sottoclasse di `Calendar`: `GregorianCalendar`. Poiché la maggior parte delle persone e praticamente tutte le società al mondo utilizzano il calendario gregoriano, tutti i dettagli sui calendari si trovano nel relativo paragrafo più avanti in questo capitolo.

Nella Tabella 10.3 sono riepilogati i metodi disponibili nella classe `Calendar`.

Tabella 10.3 *Metodi della classe Calendar.*

Metodo	*Descrizione*
Costruttori	
`Calendar()`	Crea un calendario con i valori di `TimeZone` e di `Locale` predefiniti.
`Calendar(TimeZone,Locale)`	Crea un calendario con i valori di `TimeZone` e di `Locale` specificati.
Metodi statici	
`getDefault()`	Restituisce un calendario con i valori di `TimeZone` e di `Locale` predefiniti.
`getDefault(TimeZone)`	Restituisce un calendario con il valore di `Locale` predefinito e il valore di `TimeZone` specificato.
`getDefault(Locale)`	Restituisce un calendario con il valore di `Locale` specificato e il valore di `TimeZone` predefinito.
`getDefault(TimeZone,Locale)`	Restituisce un calendario con i valori di `TimeZone` e `Locale` specificati.
`getAvailableLocales()`	Restituisce un array di tutte le località disponibili.
Metodi	
`getTime()`	Restituisce la data e l'ora.
`setTime(Date)`	Imposta la data e l'ora.
`get(byte)`	Restituisce il campo specificato.
`set(byte,int)`	Imposta il campo specificato sul valore specificato.
`set(int,int,int)`	Imposta l'anno, il mese e la data.

Tabella 10.3 *Metodi della classe Calendar. (continua)*

Metodo	*Descrizione*
`set(int,int,int,int,int)`	Imposta l'anno, il mese, la data, l'ora e i minuti.
`set(int,int,int,int,int,int)`	Imposta l'anno, il mese, la data, l'ora, i minuti e i secondi.
`clear()`	Elimina tutti i campi.
`clear(byte)`	Elimina il campo specificato.
`isSet(int)`	Restituisce `true` se il campo specificato è impostato.
`equals(Object)`	Restituisce `true` se due oggetti sono uguali.
`before(Object)`	Restituisce `true` se un oggetto è antecedente all'oggetto specificato.
`after(Object)`	Restituisce `true` se un oggetto è successivo all'oggetto specificato.
`add(byte,int)`	Aggiunge il valore specificato al campo.
`roll(byte,boolean)`	Incrementa o decrementa (in base al valore `boolean`) il campo specificato di un'unità.
`setTimeZone(TimeZone)`	Imposta il fuso orario in cui si trova un oggetto.
`setValidationMode(boolean)`	Se il valore `boolean` è impostato su `true`, sono permesse date non valide.
`getValidationMode()`	Restituisce se sono permesse o meno date non valide.
`setFirstDayOfWeek(byte)`	Imposta il primo giorno della settimana.
`getFirstDayOfWeek()`	Restituisce il primo giorno della settimana.
`setMinimumDaysInFirstWeek(byte)`	Imposta il numero di giorni necessari per definire la prima settimana del mese.
`getMinimumDaysInFirstWeek()`	Restituisce il numero di giorni necessari per definire la prima settimana del mese.
`getMinimum(byte)`	Restituisce il valore minimo possibile per il campo specificato.
`getMaximum(byte)`	Restituisce il valore massimo possibile per il campo specificato.
`getGreatestMinimum(byte)`	Restituisce il valore minimo superiore per il campo specificato.
`getLeastMaximum(byte)`	Restituisce il valore massimo inferiore per il campo specificato.
`Clone()`	Crea una copia di un oggetto.

I termini minimo superiore *e* massimo inferiore *presenti nella Tabella 10.3 potrebbero non essere chiari. Un esempio di* massimo più piccolo *è dato dalla domanda "qual è il giorno massimo più piccolo in un mese?". La risposta è 28, che corrisponde ai giorni di febbraio, dove, nella maggior parte degli anni, il giorno massimo è 28. Per il calendario gregoriano, quello più utilizzato,* `getGreatestMinimum()` *restituisce sempre lo stesso valore di* `getMinimum()`.

La classe Date

Nelle versioni di Java precedenti alla 1.1, la classe `Date` era estremamente importante per la gestione delle date e dell'ora. Tuttavia in alcune aree era debole, ad esempio nella internazionalizzazione e nella gestione delle differenze dell'ora legale tra diversi luoghi.

Con Java 1.1, la classe `Date` è stata relegata a un compito di memorizzazione dell'ora esatta; la maggior parte dei metodi della classe è stata invalidata e non dovrebbe essere più utilizzata. Tutti i metodi per convertire l'ora dalla forma binaria a quella leggibile dall'uomo e viceversa sono ora gestiti dalle classi `Calendar`, `GregorianCalendar`, `TimeZone`, `SimpleTimeZone` e `Locale`. Nel Listato 10.3, più avanti in questo capitolo, si trova un esempio del funzionamento congiunto di queste classi.

Il costruttore predefinito viene utilizzato quando sono necessarie la data e l'ora corrente. L'altro costruttore utilizza una rappresentazione dell'ora in millesimi di secondo e crea un oggetto `Date` sulla base di questa rappresentazione.

Quando una data viene convertita in una stringa da una coercizione automatica, viene utilizzato il metodo `toString()`. La stringa risultante restituita dalla funzione `toString()` segue gli standard UNIX per l'ora e la data.

Le date possono essere confrontate utilizzando i rispettivi valori UTC (il numero di secondi trascorsi dall'1 gennaio 1970) o per mezzo dei metodi `after()`, `before()` e `equals()`.

Non si dovrebbero eseguire operazioni delicate o di particolare importanza sulla base dell'ora locale del sistema operativo riflessa da Java. Nonostante l'API sia intesa per riflettere l'UTC (Coordinated Universal Time), in realtà non lo fa in modo esatto. Questo comportamento inesatto deriva dal sistema del tempo del sottostante sistema operativo. Come l'UTC, la maggior parte dei sistemi operativi moderni assume che un giorno sia uguale a 3600 secondi/ora moltiplicati per 24 ore.
Nell'UTC, tuttavia, circa una volta all'anno vi è un secondo extra, aggiunto a causa della rotazione della terra. La maggior parte degli orologi dei computer non è sufficientemente precisa da riportare questa distinzione.
Tra UTC e tempo standard dei sistemi operativi (UT/GMT) vi è una sottile differenza; uno si basa su un orologio atomico, l'altro su osservazioni astronomiche. Per tutti gli scopi pratici, questa differenza è assolutamente irrilevante.
Per ulteriori informazioni, la Sun suggerisce di visitare il sito dell'U.S Navale Observatory, in particolare il Directorate of Time all'indirizzo `http://tycho.usno.navy.mil` *e le definizioni dei diversi sistemi del tempo all'indirizzo* `http://tycho.usno.navy.mil/systime.html`.

Nella Tabella 10.4 sono riepilogati tutti i metodi non invalidati disponibili nella classe `Date`. I metodi invalidati si trovano nel manuale dell'API incluso nel JDK o nel sito Web di JavaSoft (`http://www.javasoft.com/`).

Tabella 10.4 *Metodi disponibili nell'interfaccia Date.*

Metodo	*Descrizione*
Costruttori	
`Date()`	Crea una data utilizzando la data e l'ora odierne.
`Date(long)`	Crea una data utilizzando un singolo valore UTC.
Metodi	
`after(Date)`	Restituisce `true` se la data è successiva alla data specificata.
`before(Date)`	Restituisce `true` se la data è antecedente alla data specificata.
`equals(Object)`	Restituisce `true` se la data e la data specificata sono uguali.
`getTime()`	Restituisce l'ora come singolo valore UTC.
`hashCode()`	Calcola un codice di hash per la data.
`setTime(long)`	Imposta l'ora utilizzando un singolo valore UTC.
`toString()`	Converte una data in testo utilizzando le convezioni UNIX `ctime()`.

Anche le funzioni `before()` e `after()` sono utili, poiché permettono di ottenere un'altra istanza della classe `Date` e di confrontare questa data con il valore dell'istanza che ha effettuato la chiamata.

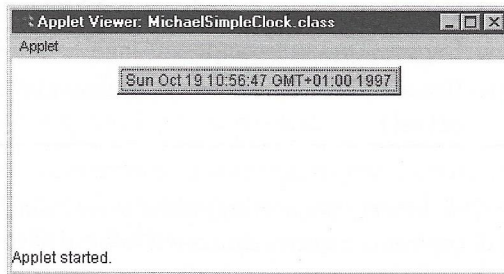
L'applet di esempio incluso nel Listato 10.2 mostra l'utilizzo della classe `Date`.

Listato 10.2 *Utilizzo della classe Date.*

```
import java.awt.*;
import java.util.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MichaelSimpleClock extends java.applet.Applet {
    Button DateButton = new Button(
        "                Fai clic qui!                ");
    public void init()  {
        ButtonListener bl = new ButtonListener(DateButton);
        add(DateButton);
     DateButton.addActionListener(bl);
    }
}

class ButtonListener implements ActionListener {
    Date TheDate = new Date();
```

```
    Button theButton;

    public ButtonListener(Button aButton) {
     theButton = aButton;
    }

    public void actionPerformed (ActionEvent e) {
           theButton.setLabel(TheDate.toString());
    }
}
```

La Figura 10.1 mostra l'applet `MichaelSimpleClock`. Per creare un orologio a tempo reale che viene aggiornato a mano a mano che cambia l'ora, è necessario includere nell'applet un ciclo in cui ogni iterazione ricrea l'istanza interna di `Date`, visualizzando ogni volta il nuovo valore per mezzo del metodo `paint()`. È inoltre necessario gestire il multithreading per fare in modo che il sistema non si blocchi durante l'esecuzione dell'applet. Il multithreading è discusso nel Capitolo 5, pertanto in questo capitolo non è stato incluso un orologio a tempo reale.

Figura 10.1 *L'applet MichaelSimpleClock.*

La classe GregorianCalendar

Fino al 1582, nel mondo occidentale si utilizzava il calendario giuliano, che era stato adottato da Giulio Cesare. Il problema di questo sistema era che aveva troppi anni bisestili (tre di troppo ogni 400 anni). Papa Gregorio XIII risolse questo problema. Inoltre l'anno giuliano iniziava il 25 marzo, mentre l'anno del calendario gregoriano inizia l'1 gennaio. Quando ci fu il cambio, l'ultimo giorno del calendario giuliano fu il 4 ottobre 1582 e il primo giorno del calendario gregoriano fu il 15 ottobre 1582. Le date comprese fra queste due non esistettero mai. Tuttavia, questo non valse per l'Inghilterra e l'America, che mantennero il calendario giuliano fino al 1752. Quando in Inghilterra e in America avvenne il cambiamento, l'ultimo giorno giuliano fu il 2 settembre 1752 e il primo giorno gregoriano fu il 14 settembre 1752. Se si lavora con date storiche, è consigliabile consultare altri testi su questo argomento. La classe `GregorianCalendar` gestisce le conversioni tra date reali e campi di date, come il mese del calendario gregoriano (il più utilizzato al mondo), tuttavia, in realtà, questa classe gestisce anche il sistema del calendario giuliano. Nella Tabella 10.5 sono riepilogati i metodi disponibili nella classe `GregorianCalendar`.

Tabella 10.5 *Metodi della classe GregorianCalendar.*

Metodo	*Descrizione*
Costruttori	
`GregorianCalendar()`	Crea un calendario gregoriano utilizzando l'ora corrente con i valori di `TimeZone` e di `Locale` predefiniti.
`GregorianCalendar(TimeZone)`	Crea un calendario gregoriano utilizzando l'ora corrente con il valore di `Locale` predefinito e il valore di `TimeZone` specificato.
`GregorianCalendar(TimeZone, Locale)`	Crea un calendario gregoriano utilizzando l'ora corrente con i valori di `TimeZone` e di `Locale` specificati.
`GregorianCalendar(int, int,int)`	Crea un calendario gregoriano all'anno, mese e data specificati con i valori di `TimeZone` e di `Locale` predefiniti.
`GregorianCalendar(int,int, int,int,int)`	Crea un calendario gregoriano all'anno, mese, data, ora e minuti specificati con i valori di `TimeZone` e di `Locale` predefiniti.
`GregorianCalendar(int,int, int,int,int,int)`	Crea un calendario gregoriano all'anno, mese, data, ora, minuti e secondi specificati con i valori di `TimeZone` e di `Locale` predefiniti.
`GregorianCalendar(Locale)`	Crea un calendario gregoriano utilizzando l'ora corrente con il valore di `TimeZone` predefinito e il valore di `Locale` specificato.
Metodi	
`setGregorianChange(Date)`	Imposta la data del passaggio dal calendario giuliano al calendario gregoriano.
`getGregorianChange()`	Restituisce la data del passaggio dal calendario giuliano al calendario gregoriano.
`isLeapYear()`	Restituisce `true` se l'anno di un oggetto è un anno bisestile.
`equals(Objects)`	Restituisce `true` se due oggetti sono uguali.
`before(Object)`	Restituisce `true` se un oggetto è antecedente alla data dell'oggetto specificato.
`after(Object)`	Restituisce `true` se un oggetto è successivo alla data dell'oggetto specificato.
`add(byte,amount)`	Aggiunge l'`amount` al campo specificato da `byte`.
`roll(byte,boolean)`	Incrementa o decrementa (in base al valore `boolean`) il campo specificato di un'unità.
`getMinimum(byte)`	Restituisce il valore minimo per il campo specificato.
`getMaximum(byte)`	Restituisce il valore massimo per il campo specificato.

Tabella 10.5 *Metodi della classe GregorianCalendar. (continua)*

Metodo	*Descrizione*
`getGreatestMinimum(byte)`	Restituisce il valore minimo superiore per il campo specificato.
`getLeastMaximum(byte)`	Restituisce il valore massimo inferiore per il campo specificato.
`clone()`	Crea una copia di un oggetto.

Il Listato 10.3 mostra l'utilizzo della classe `GregorianCalendar` e di molte altre classi del package delle utilità.

Listato 10.3 *Un programma per calcolare il giorno della settimana.*

```
import java.util.*;
public class cal {
   public static void main(String args[]) {
       int msecsInHour = 60*60*1000;
       SimpleTimeZone cst = new SimpleTimeZone(-6*msecsInHour,"CST");
       cst.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY, 2*msecsInHour);
       cst.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2*msecsInHour);
       Calendar calendar = new GregorianCalendar(cst);

       System.out.println("Day of Week: " + calendar.get(Calendar.DAY_OF_WEEK));
   }
}
```

La classe Locale

La classe `Locale` viene utilizzata per definire un *locale*, che è una combinazione di una località geografica, di una lingua e di una *variante*, che di norma è un codice specifico del browser. La classe `Locale` è essenziale nell'internazionalizzazione nel codice di Java e permette di scrivere programmi indipendenti dal paese e dalla lingua. Successivamente è possibile aggiungere comportamenti specifici per ogni paese e per ogni lingua.

I codici per le lingue ISO (International Standards Organization) sono documentati nello standard ISO-639 (`http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`), mentre i codici ISO per i paesi sono documentati nello standard ISO-3166 (`http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html`). Un esempio di codice per i paesi è `"US"` (che sta per United States, Stati Uniti); un esempio di codice per la lingua è `"EN"` (che sta per English, inglese). Alcuni esempi di varianti sono `WIN` (per Windows), `MAC` (per Macintosh) e `POSIX` (per UNIX).

Nella Tabella 10.6 sono riepilogati i metodi disponibili nella classe `Locale`.

Tabella 10.6 *Metodi disponibili con la classe Locale.*

Metodo	*Descrizione*
Costruttore	
`Locale(String, String)`	Crea una località sulla base del linguaggio e del paese specificati.
`Locale(String, String, String)`	Crea una località sulla base del linguaggio, del paese e della variante specificati.
Metodi statici	
`getDefault()`	Restituisce il valore di Locale predefinito.
`setDefault(Locale)`	Imposta il valore di Locale predefinito.
Metodi	
`getLanguage()`	Restituisce il linguaggio in uso.
`getCountry()`	Restituisce il paese in uso.
`getISO3Language()`	Restituisce il codice ISO a tre caratteri per la lingua.
`getISO3Country()`	Restituisce il codice ISO a tre caratteri per il paese.
`getVariant()`	Restituisce la variante in uso.
`toString()`	Restituisce l'oggetto come stringa.
`getDisplayLanguage()`	Restituisce la lingua da visualizzare all'utente.
`getDisplayLanguage(Locale)`	Restituisce la lingua da visualizzare all'utente sulla base del valore di `Locale` specificato.
`getDisplayCountry()`	Restituisce il paese da visualizzare all'utente.
`getDisplayCountry(Locale)`	Restituisce il paese da visualizzare all'utente sulla base del valore di `Locale` specificato.
`getDisplayVariant()`	Restituisce la variante da visualizzare all'utente.
`getDisplayVariant(Locale)`	Restituisce la variante da visualizzare all'utente sulla base del valore di `Locale` specificato.
`getDisplayName()`	Restituisce la lingua, il paese e la variante da visualizzare all'utente.
`getDisplayName(Locale)`	Restituisce la lingua, il paese e la variante da visualizzare all'utente sulla base del valore di `Locale` specificato.
`clone()`	Crea una copia di una località.
`hashCode()`	Restituisce il codice di hash per una località.
`equals(Object)`	Restituisce true se due località sono uguali.

La classe Random

Per la realizzazione di giochi e di altri tipi di programmi, è importante essere in grado di generare numeri casuali. In Java è possibile creare numeri casuali in modo efficiente ed efficace.

La classe Random implementa un tipo di dati numero pseudo-casuale che genera un flusso di numeri apparentemente casuali. Per creare una sequenza di valori diversi pseudo-casuali ogni volta che viene eseguita l'applicazione, si crea l'oggetto Random nel seguente modo:

```
Random r=new Random();
```

Questa istruzione imposta il generatore casuale con l'ora corrente. D'altra parte, si consideri la seguente istruzione:

```
Random r=new Random(326);    // Prende qualsiasi valore
```

Questa istruzione imposta il generatore casuale ogni volta con lo stesso valore, facendo in modo che venga creata la stessa sequenza di numeri pseudo-casuali ogni volta che viene eseguita l'applicazione. Il generatore può essere riprodotto in qualsiasi momento utilizzando il metodo setSeed().

Non è possibile ottenere numeri veramente casuali. Una pratica comune per simulare numeri veramente casuali nei programmi di computer consiste nell'impostare il generatore di numeri casuali con delle varianti dell'ora o della data. Se ad esempio si desidera impostare un generatore di numeri casuali con la somma dei secondi, dei minuti e delle ore, è possibile utilizzare questo codice, sufficiente per la maggior parte dei compiti:

```
int OurSeed = ADate.getSeconds() + ADate.getHours() + ADate.getMinutes();
Random = new Random(OurSeed);
```

I numeri pseudo-casuali possono essere generati utilizzando una delle seguenti funzioni: nextInt(), nextLong(), nextFloat(), nextDouble() o nextGaussian(). Le prime quattro funzioni restituiscono valori interi, lunghi, in virgola mobile e doppi. Per ulteriori informazioni sulla distribuzione gaussiana, si faccia riferimento al successivo riquadro. Il programma Random1 nel Listato 10.4 stampa cinque valori pseudo-casuali uniformemente distribuiti utilizzando queste funzioni.

Listato 10.4 *Random1.java: un programma di esempio di Random.*

```
import java.lang.Math;
import java.util.Date;
import java.util.Random;
class Random1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        int count=6;
        Random randGen=new Random();
        System.out.println("Interi casuali a distribuzione uniforme");
        for (int i=0;i<count;i++)
        System.out.print(randGen.nextInt()+" ");
```

Distribuzione gaussiana e normale

I numeri casuali uniformemente distribuiti vengono generati utilizzando un metodo lineare congruo modificato con un seme a 48 bit. I numeri casuali uniformemente distribuiti all'interno di una gamma specificata si presentano con la stessa frequenza. La classe `Random` può anche generare numeri casuali con la distribuzione gaussiana o normale. La curva di distribuzione della frequenza gaussiana viene detta anche *curva a campana*. Per ulteriori informazioni sulla curva di distribuzione della frequenza gaussiana, si rimanda a *The Art of Computer Programming*, Volume 2, di Donald Knuth.

```
        System.out.println("\n");
        System.out.println("Numeri in virgola mobile casuali a distribuzione uniforme");
        for (int i=0;i<count;i++)
        System.out.print(randGen.nextFloat()+" ");
        System.out.println("\n");
        System.out.println("Numeri in virgola mobile casuali a distribuzione gaussiana");
        for (int i=0;i<count;i++)
              System.out.print(randGen.nextGaussian()+" ");
        System.out.println("\n");
        System.out.println("Interi casuali a distribuzione uniforme [1,6]");
        for (int i=0;i<count;i++)
              System.out.print((Math.abs(randGen.nextInt())%6+1)+" ");
        System.out.println("\n");
        }
  }
```

L'output di questo programma è:

```
Interi casuali a distribuzione uniforme
1704667569 -1431446235 1024613888 438489989 710330974 -1689521238
Numeri in virgola mobile casuali a distribuzione uniforme
0.689189 0.0579988 0.0933537 0.748228 0.400992 0.222109
Numeri in virgola mobile casuali a distribuzione gaussiana
-0.201843 -0.0111578 1.63927 0.205938 -0.365471 0.626304
Interi casuali a distribuzione uniforme [1,6]
4 6 1 6 3 2
```

Se si desidera generare valori interi casuali uniformemente distribuiti all'interno di una gamma specifica, l'output di `nextInt()`, di `nextLong()` o di `nextDouble()` può essere scalato in modo da corrispondere alla gamma desiderata. Tuttavia, un approccio più semplice consiste nell'utilizzare il resto del risultato di `nextInt()` diviso per il numero di valori diversi più il primo valore della gamma. Ad esempio, se sono necessari valori da 10 a 20, è possibile utilizzare la formula `nextInt()%21+10`. Sfortunatamente, nonostante questo metodo sia più semplice rispetto alla scalatura dell'output di `nextInt()`, è sicuro solo con numeri davvero casuali. Poiché il generatore pseudo-casuale può avere diverse correlazioni non desiderate, l'operatore di modulo potrebbe non fornire risultati accettabili, ad esempio si potrebbero

ottenere solo numeri dispari. In altre parole, non è opportuno simulare operazioni delicate e molto importanti in Java, perché si potrebbero ottenere risultati non corretti.

Nella Tabella 10.7 è riepilogata l'interfaccia completa della classe `Random`.

Tabella 10.7 *I metodi disponibili nell'interfaccia Random.*

Metodo	*Descrizione*
Costruttori	
`Random()`	Crea un nuovo generatore di numeri casuali.
`Random(long)`	Crea un nuovo generatore di numeri casuali utilizzando un seme.
Metodi	
`nextDouble()`	Restituisce un numero doppio pseudo-casuale, uniformemente distribuito.
`nextFloat()`	Restituisce un numero in virgola mobile pseudo-casuale, uniformemente distribuito.
`nextGaussian()`	Restituisce un numero doppio pseudo-casuale, con distribuzione gaussiana.
`nextInt()`	Restituisce un numero intero pseudo-casuale, uniformemente distribuito.
`nextLong()`	Restituisce un numero lungo pseudo-casuale, uniformemente distribuito.
`setSeed(long)`	Imposta il seme del generatore di numeri pseudo-casuali.

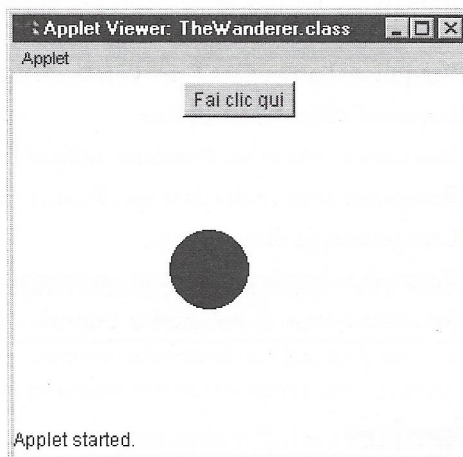
L'applet incluso nel Listato 10.5 mostra una parte di ciò che si può fare con la classe `Random`.

Listato 10.5 *Utilizzo della classe Random.*

```
import java.awt.*;
import java.util.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class TheWanderer extends java.applet.Applet {
        int xpos = 100;
        int ypos = 100;
    // Data corrente.
    Calendar C = new GregorianCalendar();
        // Pulsante di movimento
        Button theButton = new Button("Fai clic qui");
        // Generatore di numeri casuali.
        Random R;
        public void init() {
                SimpleListener simple = new SimpleListener(this);
                add(theButton);
```

```
                theButton.addActionListener(simple);
                setBackground(Color.white);
        // Generatore di numeri casuali impostato con il numero di secondi corrente.
        int seed = C.get(Calendar.SECOND);
                R = new Random(seed);
        }
    public void paint(Graphics g) {
                g.setColor(Color.black);
                g.fillOval(xpos,ypos, 50, 50);
    }

    public void move() {
                        // Spostamento.
                        xpos = xpos + (Math.abs(R.nextInt())%10-7);
                        ypos = ypos + (Math.abs(R.nextInt())%10-7);
            // Ridisegno.
                        repaint();
    }
}

class SimpleListener implements ActionListener {
        private TheWanderer theClass;

        public SimpleListener(TheWanderer aClass) {
                theClass = aClass;
            }
        public void actionPerformed (ActionEvent e) {
                theClass.move();
            }
    }
```

La Figura 10.2 mostra l'applet TheWanderer durante l'esecuzione.

Figura 10.2
L'applet TheWanderer.

La classe SimpleTimeZone

La classe `SimpleTimeZone` è un `TimeZone` semplice che può essere utilizzato con la classe `GregorianCalendar` e che assume che le regole relative al fuso orario non siano cambiate storicamente e che siano di natura semplice. Anche con queste limitazioni, questa classe può gestire le necessità della maggior parte dei programmatori. Il Listato 10.3, presentato precedentemente in questo capitolo, include un esempio dell'utilizzo di questa classe. Nella Tabella 10.8 sono riepilogati i metodi disponibili nella classe `SimpleTimeZone`.

Tabella 10.8 *Metodi disponibili nella classe SimpleTimeZone.*

Metodo	*Descrizione*
Costruttore	
`SimpleTimeZone(int, String)`	Crea un `TimeZone` con l'offset e il nome del fuso orario specificati.
`SimpleTimeZone(int, String,int,int, int, int,int,int,int,int)`	Crea un `TimeZone` con l'offset, il nome del fuso orario e l'ora di inizio e di fine dell'ora legale specificati.
Metodi	
`setStartYear(int)`	Imposta il primo anno in cui è iniziata l'ora legale.
`setStartRule(int, int,int,int)`	Imposta le regole per l'inizio dell'ora legale sulla base del mese, della settimana, del giorno della settimana e dell'ora del giorno.
`setEndRule(int,int, int,int)`	Imposta le regole per il termine dell'ora legale sulla base del mese, della settimana, del giorno della settimana e dell'ora del giorno.
`getOffset(int,int, int,int,int,int)`	Restituisce l'offset dell'ora di Greenwich (GMT) per una determinata ora.
`getRawOffset()`	Restituisce l'offset GMT per l'ora corrente.
`setRawOffset(int)`	Imposta l'offset GMT di base.
`useDaylightTime()`	Restituisce `true` se un `TimeZone` utilizza l'ora legale.
`inDaylightTime(Date)`	Restituisce `true` se alla data specificata è in vigore l'ora legale.
`clone()`	Crea una copia di un oggetto.
`hashCode()`	Restituisce il codice di hash di un oggetto.
`equals(Object)`	Restituisce `true` se un oggetto è uguale all'oggetto specificato.

La classe StringTokenizer

In questo paragrafo vengono descritte le funzionalità della classe `StringTokenizer`, che avrebbe anche potuto essere inserita tra le altre classi descritte nel Capitolo 11, in quanto è fondamentale per le funzioni di input e di output discusse in quel capitolo. La classe `StringTokenizer`

permette di suddividere una stringa in diverse stringhe più piccole, chiamate *token*. Questa classe funziona specificatamente per il cosiddetto "testo delimitato", che significa che ogni singola sottostringa è separata da un delimitatore, che può essere qualsiasi cosa, da * a YabaDaba. È sufficiente specificare che cosa deve ricercare la classe per suddividere la stringa in token.

La classe è inclusa in questo capitolo perché il suo utilizzo si dimostra utile per molte cose, dall'applet di un foglio di calcolo all'applet per un gioco.

La serie di delimitatori può essere specificata quando viene creato l'oggetto StringTokenizer o per ogni singolo token. La serie di delimitatori predefiniti è composta dai caratteri di spazio, con cui la classe trova tutte le diverse parole in una stringa e le trasforma in token. Ad esempio, il codice di StringTokenizer1 del Listato 10.6 stampa ogni parola della stringa su una riga diversa.

Listato 10.6 *StringTokenizer1.java: un programma di esempio di StringTokenizer*

```
import java.io.DataInputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;
class StringTokenizer1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        BufferedReader dis=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Inserire una frase: ");
        String s=dis.readLine();
        StringTokenizer st=new StringTokenizer(s);
        while (st.hasMoreTokens())
            System.out.println(st.nextToken());
    }
}
```

L'output di questo listato è:

```
Inserire una frase:
Quattro punti e sette
Quattro
punti
e
sette
```

Il metodo countTokens() restituisce il numero di token rimasti nella stringa utilizzando la serie corrente di delimitatori, vale a dire il numero di volte che nextToken() può essere richiamato prima di generare un'eccezione. Questo è un metodo efficiente, in quanto non crea realmente le sottostringhe che nextToken() deve generare.

Oltre a estendere la classe java.lang.Object, la classe StringTokenizer implementa l'interfaccia java.util.Enumeration.

Nella Tabella 10.9 sono riepilogati i metodi della classe StringTokenizer.

Tabella 10.9 *I metodi disponibili nell'interfaccia StringTokenizer.*

Metodo	*Descrizione*
Costruttori	
`StringTokenizer (String)`	Crea un `StringTokenizer` in base alla stringa specificata. utilizzando gli spazi come delimitatori.
`StringTokenizer (String, String)`	Crea un `StringTokenizer` in base alla stringa e alla serie di delimitatori specificati.
`StringTokenizer (String, String, boolean)`	Crea un `StringTokenizer` in base alla stringa e alla serie di delimitatori specificati; l'ultimo parametro è un valore booleano che, se `true`, indica che i delimitatori devono essere restituiti come token (se il parametro è false, i token non vengono restituiti).
Metodi	
`countTokens()`	Restituisce il numero di token rimasti nella stringa.
`hasMoreTokens()`	Restituisce `true` se esistono altri token.
`nextToken()`	Restituisce il token successivo della stringa.
`nextToken(String)`	Restituisce il token successivo, in base alla nuova serie di delimitatori specificata.
`hasMoreElements()`	Restituisce `true` se nella enumerazione esistono altri elementi.
`nextElement()`	Restituisce l'elemento successivo dell'enumerazione utilizzando la serie corrente di delimitatori.

La classe TimeZone

La classe `TimeZone` è una classe generica che rappresenta qualsiasi tipo di fuso orario. Questa classe contiene l'offset dall'ora di Greenwich (GMT) e può gestire l'ora legale. Nella Tabella 10.10 sono elencati i metodi disponibili nella classe `TimeZone`.

Tabella 10.10 *Metodi disponibili nella classe TimeZone.*

Metodo	*Descrizione*
Costruttore	
`TimeZone()`	Crea un `TimeZone`.
Metodi statici	
`getTimeZone(String)`	Restituisce l'oggetto `TimeZone` sulla base del nome del fuso orario.
`getAvailableIDs()`	Restituisce un array di tutti i nomi `TimeZone`.
`getAvailableIDs(int)`	Restituisce un array di tutti i nomi `TimeZone` a cui si applica un particolare offset.

`getDefault()`	Restituisce il `TimeZone` predefinito per un computer.
`setDefault(TimeZone)`	Imposta il `TimeZone` predefinito per un computer .
Metodi	
`getOffset(int,int, int,int,int,int)`	Restituisce l'offset GMT per il `TimeZone` locale all'ora specificata.
`getRawOffset()`	Restituisce l'offset GMT per il `TimeZone` locale all'ora corrente.
`getID()`	Restituisce il nome del `TimeZone`.
`setID(String)`	Imposta il nome del `TimeZone`.
`useDaylightTime()`	Restituisce `true` se un `TimeZone` utilizza l'ora legale.
`inDaylightTime(Date)`	Restituisce `true` se alla data specificata è in vigore l'ora legale per un `TimeZone`.
`clone()`	Crea una copia di un oggetto.

La classe Vector

Come indicato precedentemente in questo capitolo, in Java non sono inclusi gli elenchi a collegamento dinamico, le code o altre strutture di dati simili. Al loro posto, i progettisti di Java hanno previsto la classe `Vector`, che gestisce le occasioni in cui è necessario memorizzare dinamicamente gli oggetti. Naturalmente vi sono conseguenze positive e negative a questa decisione dei progettisti della Sun. Un aspetto positivo è che la classe `Vector` contribuisce a mantenere semplice il linguaggio; l'aspetto negativo principale è che, apparentemente, la classe `Vector` limita in modo significativo i programmatori nell'utilizzo di programmi più sofisticati.

In ogni caso, la classe `Vector` implementa un elenco allocato dinamicamente di oggetti e cerca di ottimizzarlo aumentando la capacità di memorizzazione dell'elenco, quando necessario, con incrementi superiori a un solo oggetto. Tipicamente, con questo meccanismo, nell'elenco vi è della capacità in eccesso. Quando questa capacità si esaurisce, l'elenco viene riallocato per aggiungere un altro blocco di oggetti alla fine. Impostando la capacità dell'oggetto `Vector` sulle dimensioni necessarie prima di inserire un grande numero di oggetti, si riduce la necessità di riallocare l'elenco per incrementarlo. È importante ricordare che, a causa di questo meccanismo, la *capacità* (gli elementi disponibili nell'oggetto `Vector`) e le *dimensioni* (il numero di elementi correntemente memorizzato nell'oggetto `Vector`) di norma non sono uguali.

Si supponga di aver creato un `Vector` con `capacityIncrement` uguale a `3`. A mano a mano che al `Vector` vengono aggiunti oggetti, viene allocato del nuovo spazio per tre oggetti alla volta. Dopo aver aggiunto cinque elementi, vi è ancora spazio per uno, senza che sia necessario allocare ulteriore memoria.

Dopo aver aggiunto il sesto elemento, non vi è più capacità in eccesso. Quando viene aggiunto il settimo elemento, viene effettuata una nuova allocazione per aggiungere altri tre elementi, dando una capacità totale di nove. Dopo aver aggiunto il settimo elemento, rimangono due elementi non utilizzati.

La capacità di memorizzazione iniziale e l'incremento della capacità possono essere entrambi specificati nel costruttore. Nonostante la capacità venga incrementata automaticamente quando necessario, è possibile utilizzare il metodo `ensureCapacity()` per aumentarla a un numero minimo specifico di elementi, mentre per ridurla al numero minimo di elementi necessari per memorizzare la quantità corrente si può utilizzare il metodo `trimToSize()`. Al `Vector` possono essere aggiunti nuovi elementi utilizzando i metodi `addElement()` e `insertElementAt()`. Gli elementi da memorizzare nel `Vector` devono derivare dal tipo `Object`; per modificarli si utilizza il metodo `setElementAt()`, mentre per eliminarli si utilizzano i metodi `removeElement()`, `removeElementAt()` e `removeAllElements()`. È possibile accedere direttamente agli elementi tramite i metodi `elementAt()`, `firstElement()` e `lastElement()` e individuarli tramite i metodi `indexOf()` e `lastIndexOf()`. Le informazioni sulle dimensioni e sulla capacità del `Vector` vengono restituite dai metodi `size()` e `capacity()`. Il metodo `setSize()` può essere utilizzato per modificare direttamente le dimensioni del `Vector`.

Ad esempio, il codice `Vector1` incluso nel Listato 10.7 crea un `Vector` di interi aggiungendo alla fine nuovi elementi, quindi lo stampa utilizzando diverse tecniche.

Listato 10.7 *Vector1.java: Un programma di esempio di Vector.*

```
import java.lang.Integer;
import java.util.Enumeration;
import java.util.Vector;
class Vector1 {
    public static void main(String args[]){
        Vector v=new Vector(10,10);
        for (int i=0;i<20;i++)
            v.addElement(new Integer(i));
        System.out.println("Vector in ordine originale con un Enumeration");
        for (Enumeration e=v.elements();e.hasMoreElements();)
            System.out.print(e.nextElement()+" ");
        System.out.println();
        System.out.println("Vector in ordine originale con elementAt");
        for (int i=0;i<v.size();i++)
            System.out.print(v.elementAt(i)+" ");
        System.out.println();
        // Stampa il vettore
        System.out.println("\nVector in ordine inverso con elementAt");
        for (int i=v.size()-1;i>=0;i++)
            System.out.print(v.elementAt(i)+" ");
        System.out.println();
        // Stampa il vettore
        System.out.println("\nVector come String");
        System.out.println(v.toString());
    }
}
```

L'output di questo programma è:

```
Vector in ordine originale con un Enumeration
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Vector in ordine originale con elementAt
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Vector in ordine inverso con elementAt
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Vector come String
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

L'espressione new `Integer()` è stata utilizzata per creare oggetti interi da memorizzare, in quanto i tipi fondamentali, quali `int` in Java non sono oggetti. Questa tecnica viene utilizzata molte volte in questo capitolo.

Si noti l'utilizzo dell'oggetto `Enumeration` per accedere agli elementi di un `Vector`. Si osservino le seguenti righe:

```
for (Enumeration e=v.elements();e.hasMoreElements();)
    System.out.print(e.nextElement()+" ");
```

Si può vedere che un oggetto `Enumeration`, che rappresenta tutti gli elementi nel `Vector`, viene creato e restituito dal metodo `elements()` di `Vector`. Con questo oggetto, il ciclo può controllare se vi sono ulteriori elementi da elaborare utilizzando il metodo `hasMoreElements()` di `Enumeration`; il ciclo può ottenere l'elemento successivo nel `Vector` utilizzando il metodo `nextElement()` di `Enumeration`.

Il programma `Vector2` incluso nel Listato 10.8 mostra alcune tecniche per l'accesso ai vettori. Prima viene generato un vettore di numeri interi casuali, quindi viene permesso all'utente di ricercare un valore specifico. La prima e l'ultima posizione del valore vengono stampate dal programma utilizzando i metodi `indexOf()` e `lastIndexOf()`.

Listato 10.8 *Vector2.java: un altro programma di esempio di Vector.*

```
import java.io.DataInputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.lang.Integer;
import java.lang.Math;
import java.util.Enumeration;
import java.util.Random;
import java.util.Vector;
class Vector2 {
    public static void main(String args[])
        throws java.io.IOException
    {
        int numElements;
        BufferedReader dis=new BufferedReader(new InputStreamReader(System.in));
        Vector v=new Vector(10,10);
        Random randGen=new Random();
        System.out.println("Quanti elementi casuali? ");
        numElements=Integer.valueOf(dis.readLine()).intValue();
        for (int i=0;i<numElements;i++)
```

```
            v.addElement(new Integer(Math.abs(
                randGen.nextInt())%numElements));
        System.out.println(v.toString());
        Integer searchValue;
        System.out.println("Quale valore cerco? ");
        searchValue=Integer.valueOf(dis.readLine());
        System.out.println("La prima occorrenza è "+
            v.indexOf(searchValue));
        System.out.println("La ultima occorrenza è "+
            v.lastIndexOf(searchValue));
    }
}
```

L'output di questo programma è:

```
Quanti elementi casuali?
10
[0, 2, 8, 4, 9, 7, 8, 6, 3, 2]
Quale valore cerco?
8
La prima occorrenza è 2
La ultima occorrenza è 6
```

Oltre a estendere la classe `java.lang.Object`, la classe `Vector` implementa l'interfaccia `java.lang.Cloneable`. Nella Tabella 10.11 sono riepilogati i metodi della classe `Vector`.

Tabella 10.11 *Le variabili e i metodi disponibili nell'interfaccia Vector.*

Variabile	*Descrizione*
`capacityIncrement`	Dimensioni delle allocazioni incrementali, in elementi.
`elementCount`	Numero di elementi in `Vector`.
`elementData`	Buffer in cui vengono memorizzati gli elementi.
Metodo	*Descrizione*
Costruttori	
`Vector()`	Crea un vettore vuoto.
`Vector(int)`	Crea un vettore vuoto con la capacità di memorizzazione specificata.
`Vector(int, int)`	Crea un vettore vuoto con la capacità di memorizzazione e gli incrementi della capacità specificati.
Metodi	
`addElement(Object)`	Aggiunge l'oggetto specificato alla fine del `Vector`.
`capacity()`	Restituisce la capacità del `Vector`.
`clone()`	Crea un clone del `Vector`.

Tabella 10.11 *Le variabili e i metodi disponibili nell'interfaccia Vector. (continua)*

Metodo	*Descrizione*
`contains(Object)`	Restituisce `true` se l'oggetto specificato è nel `Vector`.
`copyInto(Object[])`	Copia gli elementi di un vettore in un array.
`elementAt(int)`	Restituisce l'elemento nell'indice specificato.
`elements()`	Restituisce una enumerazione degli elementi.
`ensureCapacity(int)`	Garantisce che il `Vector` abbia la capacità specificata.
`firstElement()`	Restituisce il primo elemento del `Vector`.
`indexOf(Object)`	Restituisce il primo indice all'interno del Vector dell'oggetto specificato .
`indexOf(Object, int)`	Restituisce l'indice all'interno del `Vector` dell'oggetto specificato, iniziando la ricerca dall'indice specificato e procedendo verso la fine del `Vector`.
`insertElementAt(Object, int)`	Inserisce un oggetto nell'indice specificato.
`isEmpty()`	Restituisce `true` se il `Vector` è vuoto.
`lastElement()`	Restituisce l'ultimo elemento del `Vector`.
`lastIndexOf(Object)`	Restituisce l'ultimo indice all'interno del `Vector` dell'oggetto specificato.
`lastIndexOf(Object, int)`	Restituisce l'indice all'interno del `Vector` dell'oggetto specificato, iniziando la ricerca dall'indice specificato e procedendo verso l'inizio del `Vector`.
`removeAllElements()`	Rimuove tutti gli elementi del `Vector`.
`removeElement(Object)`	Rimuove l'oggetto specificato dal `Vector`.
`removeElementAt(int)`	Rimuove l'elemento con l'indice specificato.
`setElementAt(Object, int)`	Memorizza l'oggetto nell'indice specificato nel `Vector`.
`setSize(int)`	Imposta le dimensioni del `Vector`.
`size()`	Restituisce il numero di elementi nel `Vector`.
`toString()`	Converte il `Vector` in una stringa.
`trimToSize()`	Riduce la capacità del `Vector` alle dimensioni specificate.

La classe Stack

La struttura di dati dello stack è spesso fondamentale per la programmazione, ad esempio per la creazione di compilatori e per la risoluzione di complicazioni. La classe `Stack` della libreria di Java implementa uno stack di oggetti del tipo LIFO (Last In, First Out, "l'ultimo che entra è il primo a uscire"). Nonostante gli oggetti `Stack` si basino sulla classe `Vector`, che

estendono, di norma non è possibile accedere direttamente a essi, ma devono essere spinti verso l'alto e fatti uscire dallo stack. L'effetto è che i valori inseriti più recentemente sono i primi a uscire.

Il codice Stack1 nel Listato 10.9 spinge delle stringhe nella parte superiore dello stack e quindi le richiama. Le stringhe vengono infine stampate nell'ordine inverso a quello in cui sono state memorizzate.

Listato 10.9 *Stack1.java: un programma di esempio di Stack.*

```
import java.io.DataInputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Stack;
import java.util.StringTokenizer;
class Stack1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        BufferedReader dis=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Inserire una frase: ");
        String s=dis.readLine();
        StringTokenizer st=new StringTokenizer(s);
        Stack stack=new Stack();
        while (st.hasMoreTokens())
            stack.push(st.nextToken());
        while (!stack.empty())
            System.out.print((String)stack.pop()+" ");
        System.out.println();
    }
}
```

L'output di questo programma è:

```
Inserire una frase
Ma che bella giornata di sole
sole di giornata bella che Ma
```

Nonostante di norma non sia possibile accedere direttamente agli oggetti Stack, si può possibile ricercare nello stack un valore specifico utilizzando il metodo search(), che accetta un oggetto da trovare e restituisce la distanza dalla cima dello stack a cui è stato trovato l'oggetto, oppure –1 se l'oggetto non viene trovato.

Il metodo peek() restituisce l'oggetto più in alto nello stack senza eliminarlo. Se lo stack non contiene elementi, questo metodo genera un'eccezione EmptyStackException.

Nella Tabella 10.12 è riepilogata l'interfaccia completa della classe Stack.

Tabella 10.12 *I metodi disponibili nell'interfaccia di Stack.*

Metodo	*Descrizione*
Costruttore	
`Stack()`	Crea uno `Stack` vuoto.
Metodi	
`empty()`	Restituisce `true` se lo `Stack` è vuoto.
`peek()`	Restituisce l'oggetto in cima allo `Stack` senza rimuoverlo.
`pop()`	Fa uscire un elemento dallo `Stack`.
`push(Object)`	Inserisce un elemento nello `Stack`.
`search(Object)`	Trova un oggetto nello `Stack`.

La classe Dictionary

`Dictionary` è una classe astratta utilizzata come base per la classe `Hashtable`, che implementa una struttura di dati che permette di memorizzare una collezione di coppie chiave-valore. Per le chiavi e per i valori è possibile utilizzare qualsiasi tipo di oggetto. Di norma, le chiavi vengono utilizzate per trovare un particolare valore corrispondente.

Poiché la classe `Dictionary` è una classe astratta che non può essere utilizzata direttamente, gli esempi di codice in questo paragrafo non possono essere eseguiti e vengono presentati solo per spiegare lo scopo e l'utilizzo dei metodi dichiarati da questa classe. Il seguente codice, ipoteticamente, verrebbe utilizzato per creare un `Dictionary` con i seguenti valori:

```
Dictionary products = new Dictionary();
products.put(new Integer(342), "Articolo");
products.put(new Integer(124), "Oggetto");
products.put(new Integer(754), "Barra");
```

Il metodo `put()` viene utilizzato per inserire una coppia chiave-valore nel `Dictionary`. Entrambi gli argomenti devono derivare dalla classe `Object`. La chiave è il primo argomento, mentre il valore è il secondo argomento.

Un valore può essere richiamato utilizzando il metodo `get()` e una chiave specifica. Il metodo `get()` restituisce il valore `null` se non viene trovata la chiave specificata. Di seguito viene presentato un esempio:

```
String nome = products.get(new Integer(124));
if (nome != null) {
    System.out.println("Il nome del prodotto con codice 124 è " + nome);
}
```

Nonostante sia possibile richiamare un singolo oggetto con il metodo `get()`, a volte è necessario accedere a tutte le chiavi o a tutti i valori. Due metodi, `keys()` ed `elements()`, restituiscono oggetti `Enumeration` da utilizzare per accedere alle chiavi e ai valori.

Nella Tabella 10.3 è riepilogata l'interfaccia completa della classe `Dictionary`.

Tabella 10.13 *I metodi disponibili nell'interfaccia Dictionary.*

Metodo	*Descrizione*
Costruttore	
`Dictionary()`	Crea un `Dictionary` vuoto.
Metodi	
`elements()`	Restituisce una enumerazione dei valori.
`get(Object)`	Restituisce l'oggetto associato alla chiave specificata.
`isEmpty()`	Restituisce `true` se il `Dictionary` non ha elementi.
`keys()`	Restituisce una enumerazione delle chiavi.
`put(Object, Object)`	Memorizza la coppia chiave-valore specificata nel `Dictionary`.
`remove(Object)`	Elimina un elemento dal `Dictionary` sulla base della chiave.
`size()`	Restituisce il numero di elementi memorizzati.

La classe Hashtable

La struttura di dati delle tabelle di hash è molto utile per la ricerca e la manipolazione di dati. La classe `Hashtable` dovrebbe essere utilizzata se si intende memorizzare una grande quantità di dati che successivamente si dovranno ricercare. Il tempo necessario per eseguire una ricerca in una tabella di hash è decisamente inferiore a quello necessario per effettuare una ricerca in un `Vector`. Naturalmente, in caso di quantità di dati ridotte, non vi è molta differenza tra l'utilizzo di una tabella di hash o di una struttura di dati lineare, in quanto il tempo per le operazioni di supporto è di gran lunga superiore a quello richiesto per la ricerca. Nel riquadro successivo vengono fornite ulteriori informazioni sui tempi necessari per la ricerca nelle diverse classi.

L'organizzazione delle tabelle di hash si basa sulle *chiavi*, che vengono calcolate in base ai dati memorizzati. Ad esempio, se si desidera inserire diverse parole in una tabella di hash, si può basare la chiave sulla prima lettera della parola. Quando successivamente si ricerca una parola, è possibile calcolare la chiave per l'elemento ricercato. Utilizzando questa chiave, il tempo di ricerca viene drasticamente ridotto, in quanto gli elementi sono memorizzati in base al valore delle rispettive chiavi. La classe `Hashtable` implementa un meccanismo di memorizzazione delle tabelle di hash per memorizzare coppie chiave-valore. Le tabelle di hash sono progettate per individuare e richiamare in modo veloce le informazioni memorizzate utilizzando una chiave. Le chiavi e i valori possono essere oggetti di qualsiasi tipo, ma la classe dell'oggetto chiave deve implementare i metodi `hashCode()` ed `equals()`.

L'esempio `Hashtable1` nel Listato 10.10 crea un oggetto `Hashtable` e memorizza 10 coppie chiave-valore per mezzo del metodo `put()`, quindi utilizza il metodo `get()` per restituire il valore corrispondente alla chiave immessa dall'utente.

Tempi di ricerca

Il simbolo "O" maiuscola viene utilizzato per misurare i requisiti temporali nello scenario del caso peggiore per la ricerca mentre si utilizzano strutture di dati diverse. La ricerca lineare, ad esempio quella utilizzata nella classe `Vector`, è `O(n)`; la ricerca nelle tabelle di hash è `O(log n)`. Ciò significa che per numerosi oggetti è possibile risparmiare molto tempo se si utilizza una tabella di hash, perché il logaritmo di un numero è sempre inferiore al numero stesso. Se si eseguono spesso ricerche di dati, le tabelle di hash risultano molto più efficienti.

Listato 10.10 *Hashtable1.java: un programma di esempio di Hashtable.*

```
import java.io.DataInputStream;
import java.lang.Integer;
import java.lang.Math;
import java.util.Random;
import java.util.Hashtable;
class Hashtable1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        DataInputStream dis=new DataInputStream(System.in);
        int numElements=10;
        String keys[]={"Rosso","Verde","Blu","Ciano","Magenta",
            "Giallo","Nero","Arancio","Viola","Bianco"};
        Hashtable ht;
        Random randGen=new Random();
        ht=new Hashtable(numElements*2);
        for (int i=0;i<numElements;i++)
            ht.put(keys[i],new Integer(Math.abs(
                randGen.nextInt())%numElements));
        System.out.println(ht.toString());
        String keyValue;
        System.out.println("Che chiave cerco? ");
        keyValue=dis.readLine();
        Integer value=(Integer)ht.get(keyValue);
        if (value!=null) System.out.println(keyValue+" = "+value);
    }
}
```

L'output di questo programma è:

```
{Ciano=4, Bianco=0, Magenta=4, Rosso=5, Nero=3,
Verde=8, Viola=3, Arancio=4, Giallo=2, Blu=6}
Che chiave cerco?
Rosso
Rosso = 5
```

Oltre al metodo `get()`, è possibile utilizzare i metodi `contains()` e `containsKey()` per ricercare un valore o una chiave particolare. Entrambi restituiscono `true` o `false` in base al

successo della ricerca. Il metodo `contains()` deve eseguire una ricerca completa nella tabella e non è così efficiente come il metodo `containsKey()`, che può sfruttare il vantaggio del meccanismo di memorizzazione delle tabelle di hash per trovare la chiave velocemente.

Poiché le tabelle di hash devono allocare la memoria per più dati di quelli effettivamente memorizzati, esiste una misurazione, chiamata *fattore di carico, che* indica il numero di spazi di memoria utilizzati come percentuale degli spazi di memoria totali disponibili. Il fattore di carico è espresso con un valore compreso tra 0 e 100%; di norma, non dovrebbe essere superiore al 50% per poter richiamare in modo efficiente i dati in una tabella di hash. Quando in un programma si specifica il fattore di carico, si utilizza un valore compreso tra 0,0 e 1,0 per rappresentare i fattori di carico da 0 a 100%.

Le tabelle di hash possono essere create in tre modi diversi: specificando la capacità iniziale desiderata e il fattore di carico, specificando solo la capacità iniziale o senza specificare né la capacità iniziale, né il fattore di carico. Se quest'ultimo non viene specificato, le dimensioni della tabella di hash vengono aumentate quando la tabella diventa piena, o altrimenti quando viene superato il fattore di carico. Se la capacità iniziale o il fattore di carico sono inferiori o uguali a zero, i costruttori generano un'eccezione `IllegalArgumentException`.

Il metodo `clone()` può essere utilizzato per creare una copia (clone) dell'`Hashtable`, detta *copia ombra*, che significa che le chiavi e i valori non sono cloni. Questo metodo locale ridefinisce il metodo `clone()` ereditato.

L'impiego del metodo `clone()` è un'operazione relativamente costosa in termini di utilizzo di memoria e di tempo di esecuzione. Poiché la nuova `Hashtable` fa ancora riferimento agli oggetti (chiavi e valori) memorizzati nella vecchia tabella, è opportuno evitare di apportare cambiamenti che influiscono sull'`Hashtable` originale.

La classe `Hashtable` estende la classe `java.util.Dictionary` e implementa l'interfaccia `java.lang.Cloneable`. Nella Tabella 10.14 sono riepilogati i metodi della classe `Hashtable`.

La classe Properties

La classe `Properties` permette agli utenti finali di personalizzare i programmi di Java. Ad esempio, è possibile memorizzare valori come i colori di primo piano, i colori dello sfondo, i tipi di carattere predefiniti e così via e quindi fare in modo che i valori disponibili siano ricaricati. Questa possibilità è utile principalmente per le applicazioni di Java, ma è possibile implementarla anche per gli applet. Se si ha un applet che viene utilizzato regolarmente da diversi utenti, è possibile mantenere nel server un file delle proprietà per ogni utente, a cui questi accede ogni volta che carica l'applet.

La classe `Properties` è una `Hashtable`, che può essere memorizzata più volte da un flusso. Viene utilizzata per implementare proprietà permanenti e inoltre permette di creare un numero illimitato di livelli di annidamento, effettuando la ricerca in un elenco delle proprietà predefinite, se non viene trovata la proprietà richiesta. Il fatto che questa classe sia un'estensione della classe `Hashtable` comporta che tutti i metodi disponibili nella classe `Hashtable` sono disponibili anche nella classe `Properties`.

Tabella 10.14 *I metodi disponibili nell'interfaccia di Hashtable.*

Metodo	*Descrizione*
Costruttori	
`Hashtable()`	Crea una `Hashtable` vuota.
`Hashtable(int)`	Crea una `Hashtable` vuota con la capacità specificata.
`Hashtable(int, float)`	Crea una `Hashtable` vuota con la capacità e il fattore di carico specificati.
Metodi	
`clear()`	Elimina tutti gli elementi dalla `Hashtable`.
`clone()`	Crea un clone della `Hashtable`.
`contains(Object)`	Restituisce `true` se l'oggetto specificato è un elemento della `Hashtable`.
`containsKey(Object)`	Restituisce true se la `Hashtable` contiene la chiave specificata.
`elements()`	Restituisce una enumerazione dei valori della `Hashtable`.
`get(Object)`	Restituisce l'oggetto associato alla chiave specificata.
`isEmpty()`	Restituisce `true` se la `Hashtable` non ha elementi.
`keys()`	Restituisce una enumerazione delle chiavi.
`put(Object, Object)`	Memorizza la coppia chiave-valore specificata nella `Hashtable`.
`rehash()`	Inserisce il contenuto della tabella in una tabella di dimensioni maggiori.
`remove(Object)`	Elimina un elemento dalla `Hashtable` in base alla chiave dell'elemento.
`size()`	Restituisce il numero di elementi memorizzati.
`toString()`	Converte il contenuto in una stringa molto lunga.

Il programma di esempio `Properties1` incluso nel Listato 10.11 crea due elenchi di proprietà: uno è l'elenco delle proprietà predefinite e l'altro è l'elenco delle proprietà definite dall'utente. Quando viene creato l'elenco delle proprietà dell'utente, viene passato l'oggetto `Properties` predefinito. Quando si effettua la ricerca nell'elenco di proprietà dell'utente, se non viene trovato il valore della chiave, la ricerca passa nell'elenco `Properties` predefinito.

Listato 10.11 *Properties1.java: un programma di esempio di Properties.*

```
import java.io.DataInputStream;
import java.lang.Integer;
import java.util.Properties;
class Properties1 {
    public static void main(String args[])
        throws java.io.IOException
    {
```

```
        int numElements=4;
        String defaultNames[]={"Rosso","Verde","Blu","Viola"};
        int defaultValues[]={1,2,3,4};
        String userNames[]={"Rosso","Giallo","Arancio","Blu"};
        int userValues[]={100,200,300,400};
        DataInputStream dis=new DataInputStream(System.in);
        Properties defaultProps=new Properties();
        Properties userProps=new Properties(defaultProps);

        for (int i=0;i<numElements;i++){
            defaultProps.put(defaultNames[i],
                Integer.toString(defaultValues[i]));
            userProps.put(userNames[i],
                Integer.toString(userValues[i]));
        }

        System.out.println("Proprietà di default");
        defaultProps.list(System.out);

        System.out.println("\nProprietà definite dallo utente");
        userProps.list(System.out);
        String keyValue;

        System.out.println("\nChe proprietà cerco? ");
        keyValue=dis.readLine();

        System.out.println("Proprietà '"+keyValue+"' vale '"+
            userProps.getProperty(keyValue)+"'");
    }
}
```

Si noti che al posto del metodo ereditato `get()` viene utilizzato il metodo `getProperties()`; il primo effettua la ricerca solo nell'oggetto `Properties` corrente, mentre per effettuare la ricerca nell'elenco `Properties` predefinito deve essere utilizzato il metodo `getProperties()`. Una forma alternativa del metodo `getProperties()` utilizza un secondo argomento: un elenco `Properties` in cui deve essere effettuata la ricerca, al posto dell'elenco predefinito specificato quando è stato creato l'oggetto `Properties`.

Il metodo `propertyNames()` può essere utilizzato per restituire un oggetto `Enumeration`, utilizzabile per assegnare un indice a tutti i nomi delle proprietà. Questo oggetto include i nomi delle proprietà dell'elenco `Properties` predefinito.

In modo simile, il metodo `list()`, che stampa l'elenco `Properties` nel dispositivo di output standard, elenca tutte le proprietà dell'oggetto `Properties` corrente e quelle dell'oggetto `Properties` predefinito.

Gli oggetti `Properties` possono essere scritti in e letti da un dispositivo utilizzando i metodi `save()` e `load()`. Oltre al dispositivo di output o di input, il metodo `save()` ha un ulteriore argomento stringa che viene scritto all'inizio del flusso come commento.

Nella Tabella 10.15 sono riepilogati i metodi della classe `Properties`.

Tabella 10.15 *Le variabili e i metodi disponibili nell'interfaccia di Properties.*

Variabile	*Descrizione*
`defaults`	Elenco `Properties` predefinito in cui effettuare la ricerca.
Metodo	*Descrizione*
Costruttori	
`Properties()`	Crea un elenco di proprietà vuoto.
`Properties(Properties)`	Crea un elenco di proprietà vuoto con le proprietà predefinite specificate.
Metodi	
`getProperty(String)`	Restituisce una proprietà in base alla chiave.
`getProperty(String, String)`	Restituisce una proprietà in base alla chiave e alle proprietà predefinite.
`list(PrintStream)`	Stampa un elenco delle proprietà in un flusso per il debugging.
`load(InputStream)`	Legge le proprietà da un `InputStream`.
`propertyNames()`	Restituisce un elenco di tutte le chiavi.
`save(OutputStream, String)`	Scrive le chiavi in un `OutputStream`.

La classe Observable

La classe `Observable` agisce come classe di base per gli oggetti che devono essere osservati da altri oggetti che implementano l'interfaccia `Observer`. Un oggetto `Observable` può informare i suoi `Observer` di ogni modifica utilizzando il metodo `notifyObservers()`. Questo metodo esegue la notifica richiamando il metodo `update()` di tutti i suoi `Observer` e passando opzionalmente un oggetto dati che viene passato a `notifyObservers()`. Gli oggetti `Observable` possono avere qualsiasi numero di `Observer`.

Nella Tabella 10.16 è riepilogata l'interfaccia completa della classe `Observable`.

Tabella 10.16 *I metodi disponibili nell'interfaccia di Observable.*

Metodo	*Descrizione*
Costruttore	
`Observable()`	Crea un'istanza della classe `Observable`.
Metodi	
`addObserver(Observer)`	Aggiunge un `Observer` all'elenco di osservatori.
`clearChanged()`	Annulla un cambiamento osservabile.
`countObservers()`	Restituisce il numero di `Observer`.

(continua)

Tabella 10.16 *I metodi disponibili nell'interfaccia di Observable. (continua)*

Metodo	*Descrizione*
`deleteObserver(Observer)`	Elimina un `Observer` dall'elenco di osservatori.
`deleteObservers()`	Elimina tutti gli `Observer` dall'elenco di osservatori.
`hasChanged()`	Restituisce `true` se si è verificato un cambiamento osservabile.
`notifyObservers()`	Notifica tutti gli `Observer` quando si è verificato un cambiamento osservabile.
`notifyObservers(Object)`	Notifica tutti gli `Observer` di un cambiamento osservabile specifico.
`setChanged()`	Imposta un flag per indicare che si è verificato un cambiamento osservabile.

Riepilogo

In questo capitolo sono state descritte le classi che compongono il package delle utilità di Java. Questo package fornisce le implementazioni complete delle strutture di dati principali e di alcuni dei tipi di dati più utili (non i tipi numerici primitivi) necessari per i programmatori. Molti dei tipi e delle strutture di dati che si sviluppano utilizzando Java si basano sulle classi incluse nel package delle utilità. Per gli applet di dimensioni ridotte, molte di queste classi non sono necessarie; tuttavia, a mano a mano che gli applet diventano più complessi, queste classi diventano sempre più utili. In ogni caso, questo capitolo costituisce un buon punto di partenza per comprendere l'utilità di queste importanti classi di Java e per capire come utilizzarle in modo efficace.